

Processamento de Linguagens (3ºAno)

Grupo 16

Relatório do Projeto

João Magalhães
(A100740)

Jorge Rodrigues
(A101758)

Rodrigo Gomes
(A100555)

12 de maio de 2024

Resumo

Este relatório detalha o desenvolvimento de um compilador de Forth, com ênfase na ampliação da experiência em engenharia de linguagens e programação generativa. Os objetivos incluem a implementação de gramáticas independentes de contexto e tradutoras, bem como a criação de funcionalidades abrangentes, desde expressões aritméticas até condicionais e variáveis. O compilador resultante é capaz de transformar programas Forth em código nativo, otimizando a eficiência e a compactação para sistemas com recursos limitados de hardware.

Conteúdo

1	Introdução	2
2	Análise e Especificação da linguagem Forth	3
2.1	A linguagem Forth	3
2.1.1	Expressões Aritméticas	3
2.1.2	Manobras da Stack	3
2.1.3	Funções em Forth	4
2.1.4	Strings, Input e Output	4
2.1.5	Condicionais	5
2.1.6	Ciclos	6
2.1.7	Variáveis	6
2.2	Especificação dos Requisitos	6
3	Desenho do Compilador	8
3.1	Arquitetura do Compilador	8
3.2	Gramática Independente de Contexto	9
3.3	Estruturas do Compilador	11
3.3.1	Analisador Léxico	11
3.3.2	Analisador Sintático e Semântico	13
3.3.3	Controlador de Erros	18
4	Implementação e Testes	20
4.1	Alternativas, Decisões e Problemas de Implementação	20
4.2	Aspetos a melhorar	20
4.3	Testes realizados e Resultados	21
5	Conclusão	28
A	Gramática Livre de Contexto (GIC)	29
B	Código do Analisador Léxico	32
C	Código do Analisador Sintático	41

Capítulo 1

Introdução

Este relatório aborda o desenvolvimento de um compilador em Forth, uma linguagem de programação conhecida pela sua eficiência, flexibilidade e pela utilização de uma abordagem única na sintaxe e execução. O objetivo primordial deste projeto é ampliar a experiência em engenharia de linguagens e programação generativa(gramatical), especialmente no que diz respeito à escrita de gramáticas independentes de contexto (GIC) e gramáticas tradutoras (GT).

A linguagem Forth é caracterizada pelo uso de uma stack de dados para todas as operações, bem como pela notação pós-fixa (RPN), na qual os operadores são colocados após os seus operandos. Além disso, a sua natureza extensível permite que novas palavras (ou funções) sejam adicionadas de forma ágil, proporcionando uma flexibilidade incomparável na criação de programas para determinadas aplicações.

No decorrer deste trabalho, exploraremos o método da tradução dirigida pela sintaxe para desenvolver processadores de linguagens a partir de uma gramática tradutora. Utilizaremos geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

Este projeto também se propõe a implementar diversas funcionalidades, organizadas por níveis de complexidade, que abrangem desde o suporte a expressões aritméticas e criação de funções até a manipulação de condicionais, ciclos e variáveis.

Por fim, o compilador resultante será capaz de transformar programas escritos em Forth em código nativo, aproveitando-se da eficiência e compactação características, tornando-o uma escolha ideal para sistemas com recursos limitados de hardware.

Estrutura do Relatório

O presente documento está organizado em 5 capítulos. No capítulo 2 faz-se uma análise sobre a linguagem Forth de modo a poder-se especificar as entradas, resultados e formas de transformação. No capítulo 3 é especificada a estrutura utilizada para a ferramenta desenvolvida, dissecada parte a parte. No capítulo 4 fazemos referência aos resultados obtidos e aos testes utilizados para validar o projeto. No capítulo 5 referimos as conclusões retiradas do trabalho prático.

Capítulo 2

Análise e Especificação da linguagem Forth

2.1 A linguagem Forth

Como descrito no enunciado, a linguagem Forth é uma linguagem de baixo nível que opera de uma forma baseada numa *stack*. Esta linguagem possui características díspares do que é habitual nas linguagens de programação, como por exemplo, a notação pós fixa, a expansibilidade e a capacidade de tanto ser compilada como interpretada. O Forth é descrito como uma linguagem viva, onde novas tarefas e rotinas são desembaraçadamente adicionadas, transformando a tarefa de programar numa experiência dinâmica e criativa. A sua estrutura minimalista enfatiza a manipulação direta de dados através de uma *stack*, o que proporciona um ambiente altamente flexível, onde os programadores podem desenvolver soluções de forma ágil e eficiente.

2.1.1 Expressões Aritméticas

O Forth segue uma notação pós-fixa, algo que não é muito usual em linguagens de programação. Nesta notação, os operadores vêm logo a seguir aos seus operandos. Desta forma, a prioridade entre operadores deixa de ser um constrangimento, e o que passa a ser relevante é sim a ordem em que os operadores são colocados. Eis alguns exemplos de expressões aritméticas escritas na linguagem:

Exemplo 1: 2 3 +

Exemplo 2: 2 3 - 4 *

Exemplo 3: 10 1 + 2 /

Aproveitamos os exemplos para introduzir os *quickies*, um conjunto de operadores simples para operações simples, como somar mais um, ou até mesmo, um *shift* para a direita ou esquerda.

2.1.2 Manobras da Stack

Como referido, o Forth opera sobre uma só *stack*. Esta filosofia seria bastante limitadora se não existisse forma de aplicar manobras que alterassem o estado atual desta estrutura. Por esse motivo, a linguagem dispõe diversos comandos que permitem movimentos ou até mesmo remoção de elementos da *stack*. Para esta secção, escolhemos expor os seguintes comandos que consideramos mais interessantes:

1 2 SWAP	→	Stack: 2 1
1 2 DUP	→	Stack: 1 2 2
1 2 DROP	→	Stack: 1

Figura 2.1: Exemplos de manobras em Forth

2.1.3 Funções em Forth

A flexibilidade e extensibilidade da linguagem Forth surge precisamente com a versatilidade no uso de funções. De uma forma bastante única, as funções nesta linguagem operam diretamente na stack principal. Além disso, as funções podem chamar outras funções, e não há limites para que comandos possam aparecer nestas. É difícil encontrar exemplos que cubram todas as facetas das funções, mas, a propósito do relatório, selecionamos alguns exemplos ilustrativos:

```

: POW ( a b -- a^b ) 1 ROT ROT 0 DO DUP ROT * SWAP LOOP DROP ;
2 3 POW → Stack: 8
: SQUARE ( size -- square size ) DUP 0 DO DUP 0 DO ." # " LOOP CR LOOP . ;
2 SQUARE → Stack: -
    ##
    ##

```




Figura 2.2: Exemplos de funções em Forth

2.1.4 Strings, Input e Output

Para não ficar atrás de outras linguagens, e permitir que esta ferramenta possa ser utilizada em diversas situações, o Forth suporta *strings* e comandos para *I/O* que não necessitam necessariamente de ser *strings*. A sintaxe para os comandos é idêntica ao que tem sido exposto e, para as *strings*, facilmente conseguimos expor uma mensagem colocando o texto entre aspas. Dispomos dos seguintes exemplos para ilustrar alguns dos comandos mais relevantes:

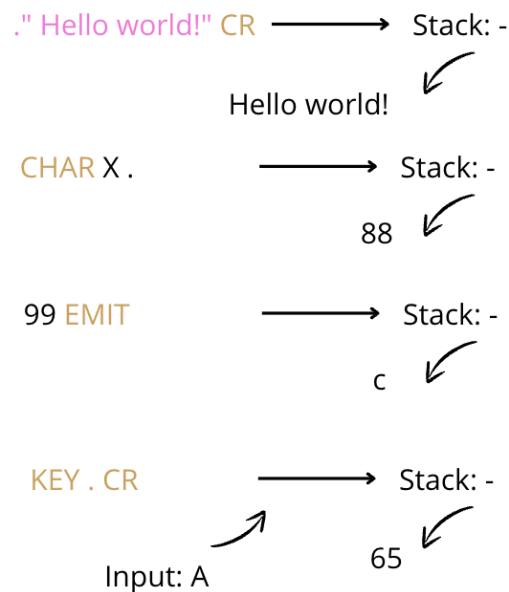


Figura 2.3: Exemplos de strings e operações I/O em Forth

2.1.5 Condicionais

As alternativas são indispensáveis para o desenvolvimento de programas. Desta forma, a linguagem Forth consta com mecanismos que permitem, através de uma condição, definir um rumo alternativo ao programa que está a ser escrito. Estas condições usam operadores lógicos, e a estrutura das alternativas segue o padrão *If, then, else* conhecido de outras linguagens.

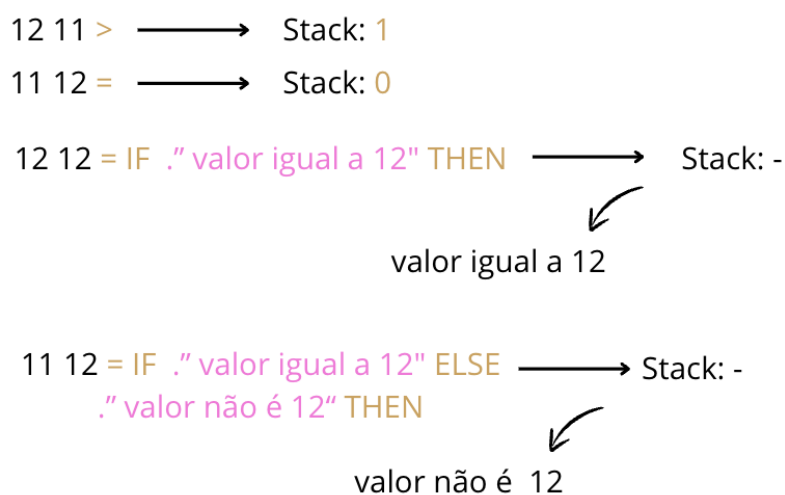


Figura 2.4: Exemplos de condições em Forth

2.1.6 Ciclos

Em muitos programas, será preciso repetir um conjunto de instruções um número fixo de vezes, ou até mesmo que uma condição se verifique. Seria complicado para um programador definir todas estas instruções manualmente, especialmente quando as repetições chegassem às centenas. Deste modo, o Forth disponibiliza ciclos. Ciclos estes que funcionam tanto para um conjunto definido de iterações como até que uma determinada condição se verifique.

5 0 DO I LOOP	→ Stack: 0 1 2 3 4
5 BEGIN 1- DUP 0= UNTIL	→ Stack: 4 3 2 1
5 0 DO I 2 +LOOP	→ Stack: 0 2 4

Figura 2.5: Exemplos de ciclos em Forth

2.1.7 Variáveis

Nesta linguagem, ainda é possível declarar variáveis. Com estas, podemos aplicar diversos comandos que permitem guardar valores, como também inserir novamente na stack o valor guardado.

VARIABLE VALOR	
10 VALOR !	→ VALOR: 10
VALOR @	→ STACK: 10
VALOR ?	→ STACK: -

10 ↙

Figura 2.6: Exemplos de uso de variáveis em Forth

2.2 Especificação dos Requisitos

Para este projeto, é sugerido no enunciado que o nosso compilador esteja devidamente preparado para compilar programas escritos em Forth. Tendo em conta os pontos descritos neste capítulos, conseguimos listar os seguintes requisitos para a nossa ferramenta:

- **Gerar código para a EWVM:** O código gerado pelo compilador têm de ser capaz de executar na máquina virtual disponibilizada pela equipa docente.
- **Análise de Expressões:** A nossa ferramenta têm de compreender expressões na notação pós-fixa e ser capaz de fazer a correspondência correta para a linguagem alvo. Além disso, deve ser abrangente nos operandos que pode interpretar.
- **Operações na Stack:** Apesar de não estar especificado no enunciado, as manobras na stack são componentes essenciais para a programação em Forth. Desta forma, queremos alcançar um comportamento semelhante com o código gerado para a máquina virtual.

- **Uso de Funções:** As funções são mecanismos que dão a flexibilidade à linguagem, logo precisamos de conseguir adequar este comportamento para a linguagem alvo.
- **Suporte a Strings e I/O:** Estas operações tornam os programas interativos com o utilizador e, por esse motivo, achamos indispensável suportar estas ações.
- **Uso de Condicionais:** As alternativas são de extrema importância, pois introduzem o poder de um programa tomar uma decisão. Por esta razão, decidimos incluir estas estruturas.
- **Ciclos:** Os ciclos permitem a criação de código limpo e com um baixo número de instruções logo suportamos os ciclos.
- **Suporte a variáveis:** As variáveis são uma componente indispensável em diversos paradigmas da programação, o que acabou por motivar a sua implementação no compilador.

Capítulo 3

Desenho do Compilador

3.1 Arquitetura do Compilador

O compilador é o engenho que habilitará a conversão do código Forth para a máquina virtual. Este sistema é geralmente dividido em camadas para facilitar o desenvolvimento e a manutenção. As camadas, tipicamente, incluem uma análise léxica, responsável por dividir o código fonte em tokens significativos, como palavras reservadas da linguagem e operadores, análise sintática, onde os tokens são agora analisados de modo a certificar que seguem a estrutura gramatical correta da linguagem Forth e outras estruturas auxiliares. Cada uma destas camadas desempenha um papel fundamental no processo de compilação, garantindo que o código fonte seja convertido de forma eficiente e precisa em código executável. O desenho e implementação do compilador deve ter em consideração os requisitos específicos da linguagem Forth, bem como as características da máquina virtual alvo, para garantir um funcionamento adequado e eficiente do sistema.

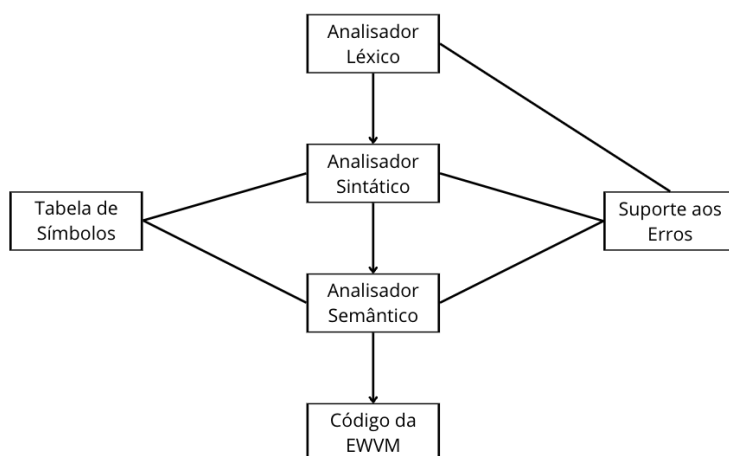


Figura 3.1: Desenho da Arquitetura compilador

3.2 Gramática Independente de Contexto

Antes de explicarmos as diferentes estruturas que compõem o compilador, queremos destacar a GIC desenvolvida para o projeto, dada a sua importância para o desenvolvimento da ferramenta. A gramática, na sua totalidade, pode ser encontrado no anexo A, pois existem produções que funcionam de forma análoga aos casos que vamos apresentar.

GIC = <T,N,S,P>

S - Options

Options - Expression
 | Function

Function - Options COLON ID Options SEMICOLON

Expression - Options Token
 | &

...

Este pequeno excerto é a parte mais importante da nossa gramática. O nosso objetivo com esta GIC é captar a essência da sintaxe da linguagem Forth. Esta linguagem é bastante peculiar na sua sintaxe. De certa forma, pode ser vista como uma lista de tokens, interpretados da esquerda para a direita. Além disso, achamos relevante separar as expressões das funções, pois as funções requerem o uso de dois tokens para delimitar as instruções da mesma. As expressões, são mais simples, e acaba por ser apenas token atrás de token.

Token - Operation
 | Quickie
 | Misk
 | Manuver
 | Logic
 | Loop
 | Condition
 | Variables
 | Elem
 | Io
 | MOD

...

Naturalmente, o símbolo não terminal "Token"encapsula todo tipo de construções sintáticas que podem aparecer num programa em Forth. Existem símbolos não terminais mais simples que derivam apenas num terminal, até casos mais complexos. Para estes casos simples, consideremos os seguintes exemplos:

Elem - NUMBER
 | STRING

```

Manuver - DUP
        | 2DUP
        | SWAP
        ...

```

```

Operation - ADD
           | MUL
           | DIV
           ...

```

```

Io - DOT
   | KEY
   | CHAR
   ...

```

...

Com estas produções, conseguimos tomar conta de três dos nossos requisitos, mais concretamente, as expressões aritméticas, com as produções derivadas do símbolo não terminal "Elem" e "Operation", as manobras na stack, graças às produções do símbolo não terminal "Manuver" e, finalmente, através das produções do símbolo não terminal "Io", as strings e ações de input e output.

```

Logic - EQUAL
       | NOTEQUAL
       | GREATER
       | LESS
       ...

```

```

Condition - IF Options ELSE Options THEN
           | IF Options THEN

```

...

Para abordar o requisito das condicionais, precisamos de montar produções que conseguissem representar a sintaxe do Forth. Para os operadores, a sintaxe é análoga aos operadores usados nas expressões aritméticas, mas para o caso dos *If statements*, a construção é bastante mais interessante. De forma sucinta, definimos que entre o token IF, ELSE e THEN, podemos encontrar Options, o que vai de acordo com o funcionamento da linguagem. Além disso, o Forth não obriga a que haja um operador lógico antes da condição, considerando sempre o último valor.

```

Loop | DO Options LOOP
     | BEGIN Options UNTIL
     | BEGIN Options WHILE Options REPEAT
     | I
     | DO Options PLUSLOOP

```

...

Os ciclos são, sintaticamente, bastante semelhantes aos *If statements*. Seguindo a sintaxe do Forth, considerando a primeira produção como exemplo, vemos que o corpo de ciclo está limitado pelos tokens DO e LOOP. O corpo do ciclo é novamente o símbolo não terminal Options, o que permite um corpo com um conjunto de instruções variado.

```
Variables - VARIABLE ID
           | ID EXCLAMATION
           | ID ATSIGN
           | ID QUESTIONMARK
           | ID
```

...

Para terminar, as variáveis também possuem uma sintaxe simples. Ao contrário da maioria das produções, consideramos que os operadores das variáveis devem suceder o respetivo ID. Isto porque conseguimos ser mais precisos com a forma como abordamos erros nas fases futuras. A última produção deste Terminal é necessária, pois representa precisamente a invocação de funções.

3.3 Estruturas do Compilador

3.3.1 Analisador Léxico

O analisador léxico foi um módulo codificado na linguagem Python, com a biblioteca *ply*, como lecionado na unidade curricular. Como referido, o objetivo desta componente é *tokenizar* o código-fonte, possibilitando o funcionamento do analisador sintático. Nesta etapa, o estudo da linguagem Forth serviu para compreender quais são os tokens que o nosso programa deve ser capaz de reconhecer.

Avançando para a implementação, de modo a manter o relatório conciso, não vamos averiguar todas as expressões regulares e tokens, assim como vamos deixar a implementação no Anexo B. Contudo existem detalhes que devem ser especificados.

Não Utilização de Literals

Optamos por não utilizar *literals*, pois achamos que compromete a longevidade da ferramenta. Ao não usar literals, se for tomada a decisão de modificar a forma que representamos uma determinada operação, as mudanças não envolvem alterações na gramática. Seguem alguns exemplos de tokens definidos:

```
tokens = [
    'NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EXP', 'LPAREN', 'RPAREN', 'DOT', 'MOD',
    ... ]
string_operands = [
    'STRING', ... 'KEY', 'EMIT'
]
reserved_words = [
    'DUP', '2DUP', 'SWAP',
    ... ]
```

```
tokens = tokens + reserved_words + logic_operands + string_operands + variables_operands
```

```
tokens = tuple(i for i in tokens)
]
```

No caso do token *EXP*, encontramos uma pequena extensão adicionada por nós à linguagem, de forma a permitir o uso de exponenciação. Caso um futuro utilizador da ferramenta achar que o *token* deve ser escrito de uma outra forma, este apenas precisa de mudar a expressão regular para o respetivo, sem se preocupar com as consequências na gramática.

Tipagem dos Tokens

Todos os tokens que passam do analisador léxico para o analisador sintático são tipados. No caso do *ply*, biblioteca de Python, este detalhe passa muitas vezes despercebido e leva a *crashes* do compilador. Desta forma, precisamos de tomar medidas no formato dos tipos dos tokens que lançamos para o *yacc*. De modo geral, o processamento dos tokens não foi necessário, contudo achamos relevante expor alguns casos para consolidar a nossa abordagem:

```
def t_PLUS(t):
    r'\+'
    return t

    ...

def t_STRING(t):
    r'\."[ ][^"]*"'
    t.value = t.value.strip(' "')
    t.value = t.value.strip(' ')
    return t

    ...

def t_NUMBER(t):
    r'[+|-]?\d+'
    t.value = int(t.value)
    return t

    ...
```

Na seleção (não ordenada) encontramos casos diferentes de processamento dos tokens. No caso do token "STRING", optamos por retirar os delimitadores, no caso dos números inteiros, optamos por converter para inteiro, pois é um tipo mais leve e, finalmente, no caso das palavras reservadas, não é preciso qualquer tipo de processamento.

Estados do Lexer

A ferramenta *ply.lex* permite ainda a definição de estados. No nosso desenho, achamos interessante considerar apenas dois estados: Estado normal e estado dentro de um comentário. Desta forma fica bastante mais simples processar os comentários que os programadores de Forth queiram deixar no código fonte.

```
states = (
    ('insidecomment','exclusive'),
)
```

Desta forma, conseguimos encapsular as palavras que são reconhecidas em comentário, das que são consideradas texto útil para o programa.

Tratamento das Palavras Reservadas

As palavras reservadas podem ser tratadas de duas formas distintas: Na primeira hipótese, assumimos que são um tipo mais geral, como um *Id* e, de seguida, verificamos numa estrutura se é uma palavra reservada, ou, alternativamente, colocámos tudo logo como token. A segunda opção pareceu-nos a mais indicada, pois faz com que estas palavras fiquem diretamente no autômato de reconhecimento gerado pelo analisador léxico. Contudo, é preciso ter em atenção a ordem de declaração dos tokens.

3.3.2 Analisador Sintático e Semântico

Na nossa arquitetura, surgem, de forma independente, o analisador sintático e o analisador semântico. Contudo, a ferramenta *yacc* da biblioteca *ply*, permite a adição de semântica, tudo no mesmo ficheiro, ao próprio analisador sintático. Essas ações semânticas são executadas conforme as regras gramaticais (já apresentadas) e são reconhecidas durante a análise sintática. Por exemplo, ao encontrar uma definição de função, o compilador pode criar uma entrada na tabela de símbolos para essa função e associar o código correspondente a essa definição.

Ao combinar a análise sintática e semântica numa única etapa, o compilador pode ser mais compacto, facilitando a sua compreensão e manutenção. Além disso, esta abordagem pode melhorar o desempenho do compilador, eliminando a necessidade de várias travessias sobre o código fonte.

De forma a manter o relatório conciso, vamos apenas dar alguns exemplos da nossa implementação, deixando o restante código no anexo C. Nesta secção, vamos tentar abranger os detalhes mais importantes da nossa implementação.

```
def p_elem1(p):
    """
    Elem          : NUMBER
    """
    p[0] = "pushi " + str(p[1]) + "\n"
    return p
...
def p_operations1(p):
    """
    Operation : PLUS
    """
    p[0] = 'ADD\n'
    return p
...
def p_manuver3(p):
    """
    Manuver      : SWAP
```

```

    """
    p[0] = "SWAP\n"
    return p
...

```

A produção `p_elem1` representa o acréscimo dos elementos numéricos à stack principal da máquina virtual. Já a produção `p_operations1` é representante das diferentes operações aritméticas. No exemplo, é adicionada ao código gerado o comando *add* da máquina virtual que soma os dois últimos valores inseridos na stack. Para terminar, as manobras são bastante parecidas às operações, onde é injetado o código referente à manobra. O caso do *swap* é dos mais simples, mas existem manobras bem mais exigentes.

```

def p_elem2(p):
    """
    Elem          : STRING
    """
    p[0] = "pushs" + ' ' + p[1] + '\n' + "writes\n"
    return p

```

```

def p_io1(p):
    """
    Io            : DOT
    """
    p[0] = "writei\n"
    return p

```

```

def p_io2(p):
    """
    Io            : KEY
    """
    p[0] = "READ\nCHRCODE\n"
    return p

```

```

def p_io3(p):
    """
    Io            : CHAR
    """
    p[0] = f'pushs"{p[1]}"\nCHRCODE\n'
    return p
...

```

De modo geral, as strings e operações de input/output foram simples e bastante diretas. Existem diferentes formas de usar strings, pois no Forth estas nunca entram diretamente na stack, ao contrário do que tem que acontecer na linguagem da máquina virtual. Para os restantes comandos, como nos exemplos nas restantes produções, a implementação é direta correspondendo apenas a um ou dois comandos da máquina virtual.


```

macro = {} #key = id, value = (string,nºcall,contains?)
parser.called_func = {} # id de função - número atual de label utilizada

def p_function1(p):
    """
    Function      : Options COLON ID Options  SEMICOLON
    """
    if macro.get(p[3]):
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(3)},
        position {p.lexpos(3)}: Function redefinition => {p[3]}\n")
    else:
        macro[p[3]] = (p[4],0,contains(p[4]))

    p[0] = p[1]
    return p

def p_variables5(p):
    """
    Variables      : ID
    """

    if macro.get(p[1]):
        m1, m2, m3 = macro[p[1]]
        m2 += 1
        if m2 == 1 or m3:
            m1_new = re_label(m1, m2, p[1])
            macro[p[1]] = (m1, m2, m3)
            p[0] = m1_new
    else:
        parser.exito = False
        parser.detected_error = 1
        errors.append(f"\tError detected in line {p.lineno(1)},
        position {p.lexpos(1)}: Unknown ID => {p[1]}\n")
        p[0] = '\n'
    return p

```

A partir da semântica adicionada na produção supracitada, conseguimos demonstrar como é que trabalhamos as funções. Dado que o Forth funciona numa só stack, acreditamos que a melhor forma de tratar funções é em estilo macro. A motivação por de trás desta decisão foi o facto de todas as instruções no Forth operarem numa só stack. O mecanismo de chamada de funções na *EWVM* utiliza stacks distintas, o que não promove o resultado esperado. Quando a função é chamada, pode ser necessário fazer atualização em labels já definidas.

```

parser.if_counter = 1

```

```

def p_condition1(p):
    """
    Condition      : IF Options ELSE Options THEN
    """
    c = parser.if_counter
    parser.if_counter += 1

    p[0] = "jz " + "ELSE" + str(c) + "\n" + p[2] + "jump ENDIF" + str(c) + "\n" + "ELSE"
    + str(c) + ":\n" + p[4] + "ENDIF" + str(c) + ":\n"
    return p

```

Para conseguirmos o comportamento esperado pelo código Forth no código da máquina virtual, no caso das condicionais, é necessário a introdução de *labels*. As *labels*, na linguagem da máquina virtual, servem como marcação para posteriormente conseguirmos saltar instruções dependendo de uma determinada condição. Para este efeito, é preciso que todas as etiquetas sejam únicas e, por isso, é preciso manter uma contagem de etiquetas já atribuídas. A geração de código é simples: caso a condição não se verifique, saltamos para a label “ELSEX:”, e caso seja verificada, no final desse pequeno corpo, fazemos o salto para o “ENDIFX:”, onde X é o número da label.

```

parser.loop_label = 0

```

```

def p_loop1(p):
    """
    Loop           : DO Options LOOP
    """
    v = parser.var_counter
    parser.var_counter += 2

    loop = "storeg " + str(v+1) + "\n" + "storeg " + str(v) + "\n" + "LOOPLABEL" +
    str(parser.loop_label) + ":\n" + p[2] + "pushg " + str(v+1) + "\npushi 1\nADD
    \nDUP 1\nstoreg" + str(v+1) + "\npushg " + str(v) + "\nSUPEQ\njz LOOPLABEL" +
    str(parser.loop_label) + "\n"

    p[0] = handle_loop_var(loop, v+1)
    parser.loop_label += 1
    return p

```

```

def p_loop2(p):
    """
    Loop           : BEGIN Options UNTIL
    """

    p[0] = "LOOPLABEL" + str(parser.loop_label) + ":\n" + p[2] +
    "jz LOOPLABEL" + str(parser.loop_label) + "\n"
    parser.loop_label += 1
    return p

```

Existem dois tipos de ciclo. Os ciclos limitados por número de iteração que requerem o uso de memória de forma a manter o número total de iterações, assim como a iteração atual e os ciclos condicionais que repetem o corpo até uma determinada condição ser verdadeira. Para este efeito, usamos uma etiqueta, que delimita o topo do ciclo e, assim que as instruções do corpo terminarem, testamos a iteração atual de forma a saber se já está na altura de sair do ciclo.

```

parser.var_counter = 0
def p_variables1(p):
    """
    Variables      : VARIABLE ID
    """
    if p[2] in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(2)},
        position {p.lexpos(2)}: Duplicate ID => {p[2]}\n")
        p[0] = '\n'
    else:
        parser.symbol_table[p[2]] = (parser.var_counter,1)
        parser.var_counter += 1
        p[0] = ""
    return p

def p_variables2(p):
    """
    Variables      : ID EXCLAMATION
    """
    if p[1] not in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(1)},
        position {p.lexpos(1)}: Unknown ID => {p[1]}\n")
        p[0] = '\n'
    else:
        pos,t = parser.symbol_table[p[1]]
        p[0] = "storeg " + str(pos) + "\n"
    return p

def p_variables3(p):
    """
    Variables      : ID ATSIGN
    """
    if p[1] not in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(1)},
        position {p.lexpos(1)}: Unknown ID => {p[1]}\n")
        p[0] = '\n'
    else:

```

```

    pos,t = parser.symbol_table[p[1]]
    p[0] = "pushg " + str(pos) + "\n"

    return p

```

As variáveis necessitam do uso de uma estrutura conhecida por tabela de símbolos. Nesta estrutura, mantém-se informação relativa ao mapeamento da variável com a posição na memória, como também o espaço usado em memória. Na produção `p_variables1`, é o momento onde essa alocação é feita. A partir daí, as outras produções expostas servem-se dessa estrutura para aceder com precisão ao conteúdo ou até mesmo à posição da variável.

```

def p_s1(p):
    """
    S      : Options
    """
    p[0] = ''
    if bad_loop_var(p[1]):
        parser.exito = False
        errors.append(f"\tError: Bad usage of I =>\n")
        p[0] = '\n'
    if parser.exito:
        for i in range(parser.var_counter):
            p[0] += "pushi 0\n"
        p[0] += 'START\n' + p[1] + 'STOP\n'

    return p

```

Para terminar, gostávamos de mostrar como é que o programa é fechado logo após o *parsing*. Assumindo que não aconteceram erros, as instruções são agora limitadas pelos comandos "Start" e "Stop", algo necessário para sintaxe da *EWVM*. Além disso, são alocadas todos os espaços na memória necessários para o funcionamento correto do programa.

3.3.3 Controlador de Erros

A última estrutura que merece destaque no nosso engenho é o controlador de erros. Na verdade, o controlo de erros é incorporado no *yacc*, de forma semelhante à geração de código e à tabela de símbolos. No entanto, existem aspetos que gostávamos de realçar, e expor alguns exemplos de controlo de erros utilizados neste compilador.

```

errors = []
parser.exito = True

```

A nossa primeira decisão foi relativa à terminação do compilador em caso de erro. No nosso entender, é mais benéfico e informativo para quem está a escrever o código fonte que todos os erros sejam corretamente

apontados, na vez de só o primeiro. Daí a necessidade de uma estrutura que segure os diferentes erros encontrados durante o *parsing*.

Atendendo ao funcionamento da linguagem Forth, os erros normalmente enquadram-se dentro dos seguintes pontos:

- **Redeclaração de Variáveis ou Funções:** Um erro frequente é tentar atribuir a um identificador um novo significado, sendo que este já possui um contexto no programa. Este erro pode ser tratado nas seguintes produções:

```
def p_function1(p):
    """
    Function      : Options COLON ID Options  SEMICOLON
    """
    if macro.get(p[3]):
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(3)},
        position {p.lexpos(3)}: Function redefinition => {p[3]}\n")
    ...
```

- **Identificador não declarado:** Existem situações onde aparecem identificadores que não lhes foram atribuído nenhum significado. Nestes casos, o compilador nem sequer consegue fazer distinção entre variável ou função.

```
def p_variables4(p):
    """
    Variables      : ID QUESTIONMARK
    """
    if p[1] not in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(1)},
        position {p.lexpos(1)}: Unknown ID => {p[1]}\n")
        p[0] = '\n'
    ...
```

- **Mau uso de Palavra Reservada:** O último tipo de erros que devem ser considerados em compilação são o mau uso de palavras reservadas, mais concertamento, erros sintáticos. Estas situações acontecem quando um token reconhecido pelo analisador léxico não permite qualquer tipo de transições de estado disponíveis para o estado atual.

```
def p_variables4(p):
    """
    Variables      : ID QUESTIONMARK
    """
    if p[1] not in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(1)},
        position {p.lexpos(1)}: Unknown ID => {p[1]}\n")
        p[0] = '\n'
    ...
```

Capítulo 4

Implementação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

Tendo já sido mencionado, é de grande importância reafirmar a nossa decisão relativamente à forma como as funções se encontram implementadas, dado que essa alternativa se demonstrou condicionante em certas áreas do desenvolvimento deste projeto. Tendo um certo respeito pela implementação original da linguagem Forth, optamos por seguir com uma implementação que depende somente de uma stack, atribuindo às funções conotação de macros, no sentido em que, sendo definida uma função, as instruções VM resultantes seriam guardadas e, conseqüentemente, enviadas sempre que a mesma fosse chamada.

Rapidamente nos demos conta de que este método nos traria algumas preocupações extra. Aquando da realização de testes a meio do processo de implementação do analisador sintático, deparamo-nos com erros quando chamávamos uma função mais que uma vez, se esta contivesse labels. O porquê de tal acontecimento, embora nos tivesse passado despercebido ao início, é bastante fácil de compreender: Sendo que as labels tinham um nome fixo numa função e que as instruções depositadas na VM relativas a uma função eram sempre as mesmas, ocorria conflito entre as labels, que teriam o mesmo nome. De forma a resolver este problema, implementamos uma função que realiza um "re-label", sendo que as labels de uma função passaram a conter o número de chamamento (1 na primeira vez, 2 na segunda...) e o nome da função em si. Mais à frente, notamos que a VM não aceita labels com caracteres especiais, os quais, por sua vez, são perfeitamente aceites pelo Forth na definição do nome de funções. De modo a manter esta última implementação, as labels passam ainda por um processo de remoção desses mesmos caracteres.

A deteção de problemas deste género foi deveras facilitada pela implementação do que achamos ser um bom sistema de deteção de erros, a ser explorado mais à frente, na secção de testes e resultados.

4.2 Aspetos a melhorar

Desde o início da planificação do nosso projeto que o grupo definiu a reutilização de posições de memória como algo de elevada prioridade. Assim sendo, quando terminamos o código base, por assim dizer, focamo-nos nessa implementação, a qual seria possível, como já tínhamos visto desde o início, através do uso de uma AST (abstract syntax tree). Existindo uma necessidade de utilização de variável acima de outra na árvore, seriam utilizadas uma posição de memória para cada, caso contrário, poder-se-ia reutilizar uma posição para a outra. O que o grupo não estudou aprofundadamente no início foi o método de implementação da AST.

Assim sendo, quando tomamos conhecimento de que teríamos de fazer uma reestruturação completa do projeto, não possuíamos tempo suficiente para o fazer, revelando-se, portanto, a reutilização de posições de memória uma funcionalidade por implementar no nosso código final.

Pelos mesmos dois motivos, método de implementação utilizado e tempo, a recursividade de funções também não se encontra implementada.

4.3 Testes realizados e Resultados

Mostram-se, de seguida, imagens relativas a testes realizados, em que introduzimos linguagem Forth no nosso programa, e aos resultados obtidos, instruções para a EWVM. De forma a não ficar uma análise bastante extensiva, são demonstrados somente excertos dos ficheiros resultado. No entanto, os ficheiros completos podem ser consultados na pasta do projeto:

Testes Aritméticos:

```
30 5 - .      30 5 / .      30 5 * .      30 5 + 7 / .      11 5 /MOD . .      1 3 ABS .
3 negate .    1 9 MIN .      3 2 Max .      1 2+                1 1+                3 1-
2 2-          2 2*          4 2/
```

Resultados Testes Aritméticos:

```
pushi 0
START
pushi 30
pushi 5
SUB
...
pushsp
LOAD -1
pushsp
LOAD -1
MOD
STOP
```

Testes Comentário:

```
1 2 + ( isto e um teste -- bla )
3 \ tambem e valido
```

Resultados Testes Comentário:

```
START
pushi 1
pushi 2
ADD
pushi 3
STOP
```

Testes Condições:

```
2 3 2dup > if drop else swap drop then
2 3 2dup = if swap . then .
```

Resultados Testes Condições:

```
START
pushi 2
pushi 3
pushsp
LOAD -1
pushsp
LOAD -1
SUP
jz ELSE1
POP 1
jump ENDIF1
ELSE1:
SWAP
POP 1
ENDIF1:
...
STOP
```

Testes Funções:

```
: AVERAGE ( a b -- avg ) + 2/ ;
: CENA ;
: double ( n -- n*2 ) 2 * ;
5 double .
: is-even? ( n -- flag ) 2 MOD 0= ;
6 is-even? .
: even-or-double ( n -- n|2n )
  dup is-even? IF
    double ELSE dup THEN ;
6 even-or-double .
```

Resultados Testes Funções:

```
START
pushi 5
pushi 2
MUL
...
EQUAL
jz evenordouble1ELSE1
pushi 2
MUL
jump evenordouble1ENDIF1
evenordouble1ELSE1:
```



```
DUP 1
evenordouble1ENDIF1:
writei
STOP
```

Testes I/O:

```
5 . key char ola char + cr space 5 spaces 97 emit
```

Resultados Testes I/O:

```
pushi 0
START
pushi 5
writei
READ
CHRCODE
pushs"o"
CHRCODE
...
DUP 1
storeg 0
pushi 0
EQUAL
jz LOOPLABEL0
pushi 97
WRITECHR
STOP
```

Testes Lógicos:

```
5 5 = . 5 5 <> . 5 3 > . 3 5 < . 0 0= . 0 0< . 1 0> . FALSE . TRUE .
```

Resultados Testes Lógicos:

```
START
pushi 5
pushi 5
EQUAL
...
pushi 0
EQUAL
writei
pushi 0
pushi 0
INF
writei
pushi 1
```

```

pushi 0
SUP
...
writei
STOP

```

Testes Loop:

```

10 3 DO
    I .
LOOP
cr
0 BEGIN 1 + DUP . DUP 5 = UNTIL
cr
12 BEGIN
    DUP . 1+ DUP 15 <
WHILE
    20 .
REPEAT
cr
10 0 DO I . 2 +LOOP

```

Resultados Testes Loop:

```

pushi 0
pushi 0
pushi 0
pushi 0
START
...
storeg 1
pushg 0
SUPEQ
jz LOOPLABEL0
WRITELN
pushi 0
LOOPLABEL1:
pushi 1
ADD
DUP 1
writei
DUP 1
pushi 5
EQUAL
...
jz LOOPLABEL4
STOP

```

Testes Manuver:

5 dup . 5 3 2dup 5 3 swap . 1 2 3 rot . . . 5 3 over . 5 drop .

Resultados Testes Manuver:

```
pushi 0
START
pushi 5
...
writei
pushi 5
pushi 3
SWAP
writei
pushi 1
pushi 2
pushi 3
storeg 0
SWAP
pushg 0
SWAP
...
pushi 5
POP 1
writei
STOP
```

Testes Misk:

5 dup . 5 3 2dup 5 3 swap . 1 2 3 rot . . . 5 3 over . 5 drop .

Resultados Testes Misk:

```
START
pushi -5
DUP 1
...
pushi 3
pushsp
LOAD -1
pushsp
LOAD -1
INF
jz ELSE2
POP 1
jump ENDIF2
ELSE2:
SWAP
POP 1
ENDIF2:
```

```

writei
pushi 5
...
ENDIF3:
writei
STOP

```

Testes String:

```

: SPROUTS ." Miniature vegetables." ;
: MENU CR SPROUTS CR ;
MENU

```

Resultados Testes String:

```

START
WRITELN
pushs"Miniature vegetables"
writes
WRITELN
STOP

```

Testes Variáveis:

```

variable my-var
5 my-var !
my-var ?
my-var @

```

Resultados Testes Variáveis:

```

pushi 0
START
pushi 5
storeg 0
pushg 0
writei
pushg 0
STOP

```

É ainda de mencionar que todos estes testes se encontram corretos de acordo com a linguagem Forth e foram verificados na máquina virtual, EWVM, tendo obtido uma taxa de sucesso de 100%. Apresenta-se de seguida uma mensagem de erro, produzida pelo sistema por nós desenhado, de modo a rematar o módulo de testes com a ferramenta que nos permitiu aprimorar o código produzido e o nível de exigência dos respetivos testes. Através das mensagens, conseguimos ver o tipo de erro e a sua localização, bem como todos os erros encontrados e não somente o primeiro que fizesse o programa "estourar". Este exemplo em específico, levou-nos não só a corrigir expressões regulares no Lexer, mas também o problema de labels já mencionado.

```
error_file.txt X
error_file.txt
1 11 ERROR(S) FOUND:
2
3 Syntactic error detected => Bad use of reserved words
4 Error detected in line 1, position 43: Unknown ID => uble
5 Syntactic error detected => Bad use of reserved words
6 Error detected in line 1, position 70: Unknown ID => uble
7 Syntactic error detected => Bad use of reserved words
8 Syntactic error detected => Bad use of reserved words
9 Error detected in line 1, position 113: Unknown ID => is-even
10 Error detected in line 1, position 153: Unknown ID => is-even
11 Error detected in line 1, position 166: Unknown ID => ubleELSEdupTHEN
12 Syntactic error detected => Bad use of reserved words
13 Error detected in line 1, position 185: Unknown ID => even-or-double
14
```

Figura 4.1: Mensagens de erro produzidas pelo programa, face a input não aceitado

Capítulo 5

Conclusão

Este projeto representou uma imersão profunda no desenvolvimento de um compilador para a linguagem Forth, conhecida pela sua eficiência, flexibilidade e abordagem única na sintaxe e execução. O objetivo principal foi ampliar a nossa experiência em engenharia de linguagens e programação generativa, especialmente na escrita de gramáticas independentes de contexto (GIC) e gramáticas tradutoras (GT).

Esta linguagem destaca-se pelo uso de uma stack de dados para todas as operações, bem como pela notação pós-fixa (RPN), características que desafiaram e enriqueceram a nossa compreensão de paradigmas de programação alternativos. Além disso, a sua natureza extensível permite a adição ágil de novas palavras, proporcionando uma flexibilidade incomparável na criação de programas para diversas aplicações.

Implementamos diversas funcionalidades, organizadas por níveis de complexidade, que abrangem desde o suporte a expressões aritméticas e criação de funções até a manipulação de condicionais, ciclos e variáveis, resultando num produto final capaz de transformar programas escritos em Forth em código nativo.

Este projeto proporcionou-nos uma experiência valiosa de aprendizagem e desenvolvimento e estamos confiantes de que as lições aprendidas servirão em projetos futuros e contribuirão para nossa contínua evolução.

Apêndice A

Gramática Livre de Contexto (GIC)

Lista-se a seguir a GIC utilizada no projeto.

GIC = $\langle T, N, S, P \rangle$

S - Options

Options - Expression
| Function

Function - Options COLON ID Options SEMICOLON

Expression - Options Token
| &

Token - Operation
| Quickie
| Misk
| Manuver
| Logic
| Loop
| Condition
| Variables
| Elem
| Io

Operation - ADD
| MUL
| DIV
| SUB
| EXP
| MOD
| DIVMOD

Loop | DO Options LOOP

```

| BEGIN Options UNTIL
| BEGIN Options WHILE Options REPEAT
| I
| DO Options PLUSLOOP

Condition - IF Options ELSE Options THEN
| IF Options THEN

Quickie - QUICKADDONE
| QUICKADDTWO
| QUICKSUBONE
| QUICKSUBTWO
| QUICKMULTWO
| QUICKDIVTWO

Logic - EQUAL
| NOTEQUAL
| GREATER
| LESS
| ZEROEQUALS
| ZEROLESS
| ZEROGREATER
| FALSE
| TRUE

Misk - ABS
| NEGATE
| MIN
| MAX

Elem - NUMBER
| STRING

Manuver - DUP
| 2DUP
| SWAP
| ROT
| OVER
| DROP

Io - DOT
| KEY
| CHAR
| CR
| SPACE
| SPACES
| EMIT

```



```
Variables - VARIABLE ID
          | ID EXCLAMATION
          | ID ATSIGN
          | ID QUESTIONMARK
          | ID
```

Apêndice B

Código do Analisador Léxico

Lista-se a seguir o código que foi desenvolvido para o Analisador Léxico.

```
import ply.lex as lex
import sys
# List of token names.    This is always required
tokens = [
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'EXP',
    'LPAREN',
    'RPAREN',
    'DOT',
    'DIVMOD',
    'MOD',
    'MIN',
    'MAX',
    'ABS',
    'NEGATE',
    'QUICKADDONE',
    'QUICKADDTWO',
    'QUICKSUBONE',
    'QUICKSUBTWO',
    'QUICKMULTWO',
    'QUICKDIVTWO',
    'SEMICOLON',
    'COLON',
    'ID',
    'BEGINCOMMENT',
    'WHITESPACE',
    'NEWLINE'
]
```

```

logic_operands = [
    'EQUAL',
    'NOTEQUAL',
    'GREATER',
    'LESS',
    'ZEROEQUALS',
    'ZEROLESS',
    'ZEROGREATER',
    'INVERT',
    'FALSE',
    'TRUE'
]

```

```

string_operands = [

    'STRING',
    'KEY',
    'SPACE',
    'SPACES',
    'CHAR',
    'CR',
    'EMIT'
]

```

```

reserved_words = [
    'DUP',
    '2DUP',
    'SWAP',
    'OVER',
    'ROT',
    'DROP',
    'IF',
    'THEN',
    'ELSE',
    'DO',
    'LOOP',
    'PLUSLOOP',
    'BEGIN',
    'UNTIL',
    'WHILE',
    'REPEAT',
    'VARIABLE',
    'I'
]

```

```

variables_operands = [

```

```

    'ATSIGN',
    'EXCLAMATION',
    'QUESTIONMARK'
]

tokens = tokens + reserved_words + logic_operands + string_operands + variables_operands

tokens = tuple(i for i in tokens)

states = (
    ('insidecomment', 'exclusive'),
)

def t_insidecomment_RPAREN(t):
    r'\)[ ]*'
    t.lexer.begin('INITIAL')

def t_insidecomment_ignore_BEGINCOMMENT(t):
    r'[\n)]'
    pass

t_insidecomment_ignore = ''

# Error handling rule
def t_insidecomment_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Regular expression rules for the comment state
def t_WHITESPACE(t):
    r'[ ]+'

def t_ATSIGN(t):
    r'\@'
    return t

def t_EXCLAMATION(t):
    r'\!'
    return t

def t_QUESTIONMARK(t):
    r'\?'
    return t

def t_VARIABLE(t):
    r'(?i:variable)'

```

```

    return t

def t_DO(t):
    r'(?i:do\b)'
    return t

def t_PLUSLOOP(t):
    r'\+(?i:loop)'
    return t

def t_LOOP(t):
    r'(?i:loop)'
    return t

def t_BEGIN(t):
    r'(?i:begin)'
    return t

def t_UNTIL(t):
    r'(?i:until)'
    return t

def t_WHILE(t):
    r'(?i:while)'
    return t

def t_REPEAT(t):
    r'(?i:repeat)'
    return t

def t_IF(t):
    r'(?i:if)'
    return t

def t_ELSE(t):
    r'(?i:else)'
    return t

def t_THEN(t):
    r'(?i:then)'
    return t

def t_SWAP(t):
    r'(?i:swap)'
    return t

def t_DUP(t):
    r'(?i:dup)'

```

```

    return t

def t_2DUP(t):
    r'2(?i:dup)'
    return t

def t_OVER(t):
    r'(?i:over)'
    return t

def t_ROT(t):
    r'(?i:rot)'
    return t

def t_DROP(t):
    r'(?i:drop)'
    return t

def t_MIN(t):
    r'(?i:min)'
    return t

def t_MAX(t):
    r'(?i:max)'
    return t

def t_ABS(t):
    r'(?i:abs)'
    return t

def t_NEGATE(t):
    r'(?i:negate)'
    return t

# Palavras reservadas ficam em cima desta linha

def t_QUICKADDONE(t):
    r'1\+'
    return t

def t_QUICKADDTWO(t):
    r'2\+'
    return t

def t_QUICKSUBONE(t):
    r'1\-'
    return t

```

```

def t_QUICKSUBTWO(t):
    r'2\-'
    return t

def t_QUICKMULTWO(t):
    r'2\*'
    return t

def t_QUICKDIVTWO(t):
    r'2\/'
    return t

# Quickies em cima

def t_KEY(t):
    r'(?i:key)'
    return t

def t_CHAR(t):
    r'(?i:char\s+\S+)'
    t.value = t.value.split()[1][0]
    return t

def t_EMIT(t):
    r'(?i:emit)'
    return t

def t_SPACES(t):
    r'(?i:spaces)'
    return t

def t_SPACE(t):
    r'(?i:space)'
    return t

def t_CR(t):
    r'(?i:cr)'
    return t

def t_STRING(t):
    r'\."[ ]["]*"'
    t.value = t.value.strip(' "')
    t.value = t.value.strip(' ')
    return t

# String definiton em cima

```

```

def t_EQUAL(t):
    r'\='
    return t
def t_NOTEQUAL(t):
    r'\<\>'
    return t

def t_LESS(t):
    r'\<'
    return t

def t_GREATER(t):
    r'\>'
    return t

def t_ZEROEQUALS(t):
    r'\0\='
    return t

def t_ZEROLESS(t):
    r'\0\<'
    return t

def t_ZEROGREATER(t):
    r'\0\>'
    return t

def t_INVERT(t):
    r'(?i:invert)'
    return t

def t_TRUE(t):
    r'(?i:true)'
    return t

def t_FALSE(t):
    r'(?i:false)'
    return t

# Condições em cima

def t_DOT(t):
    r'\.'
    return t

def t_NUMBER(t):
    r'[+\-]?\d+'
    t.value = int(t.value)

```



```

    return t

def t_LPAREN(t):
    r'\( '
    t.lexer.begin('insidecomment')

def t_SEMICOLON(t):
    r'\;'
    return t

def t_COLON(t):
    r'\:'
    return t

def t_PLUS(t):
    r'\+'
    return t

def t_MINUS(t):
    r'\-'
    return t

def t_TIMES(t):
    r'\*'
    return t

def t_DIVMOD(t):
    r'(?i:\/mod)'
    return t

def t_MOD(t):
    r'(?i:mod)'
    return t

def t_DIVIDE(t):
    r'\/'
    return t

def t_EXP(t):
    r'(?i:exp)'
    return t

def t_I(t):
    r'(?i:i\b)'
    return t

def t_ID(t):
    r'[a-zA-Z_.(){}+\-*\/][a-zA-Z_.(){}1-9+\-*\/:;]*'

```

```

    return t

def t_BEGINCOMMENT(t):
    r'\\.*'

# Define a rule so we can track line numbers
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore = '\t\r'

def t_error(t):
    line_number = t.lexer.lineno
    illegal_char = t.value[0]
    print(f"Illegal character '{illegal_char}' at line {line_number}")
    exit(1)

lexer = lex.lex()

```

Apêndice C

Código do Analisador Sintático

Lista-se a seguir o código que foi desenvolvido para o Analisador Sintático.

```
import sys
import ply.yacc as yacc
from analex import tokens
import re

macro = {} #key = id, value = (string,nºcall,contains?)
errors = []

def re_label(body, num, id):
    patterns = [r'(LOOPLABEL)',r'(ELSE)',r'(ENDIF)']
    for pattern in patterns:
        body = re.sub(pattern, lambda match: id + str(num) + match.group(1), body)
    return body

def contains(func_body):
    pattern = r'\bENDIF || \bLOOPLABEL'

    if re.match(pattern, func_body):
        return 1
    else:
        return 0

def bad_loop_var(body):
    pattern = r'loopvar'

    if re.search(pattern,body):
        return True
    else:
        return False

def handle_loop_var(loop, v):
    pattern1 = r'loopvar'
```

```

body = re.sub(pattern1, "pushg " + str(v), loop)
return body

def p_s1(p):
    """
    S      : Options
    """
    p[0] = ''
    if bad_loop_var(p[1]):
        parser.exito = False
        errors.append(f"\tError: Bad usage of I =>\n")
        p[0] = '\n'
    if parser.exito:
        for i in range(parser.var_counter):
            p[0] += "pushi 0\n"
        p[0] += 'START\n' + p[1] + 'STOP\n'

    return p

def p_options1(p):
    """
    Options      : Expression
    """
    p[0] = p[1]
    return p

def p_options2(p):
    """
    Options      : Function
    """
    p[0] = p[1]
    return p

def p_function1(p):
    """
    Function      : Options COLON ID Options SEMICOLON
    """
    if macro.get(p[3]):
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(3)}, position {p.lexpos(3)}: Function")
    else:
        macro[p[3]] = (p[4], 0, contains(p[4]))

    p[0] = p[1]
    return p

```

```
def p_expression1(p):
    """
    Expression : Options Token
    """
    p[0] = p[1] + p[2]
    #print(p[0], end='')
    return p
```

```
def p_expression2(p):
    """
    Expression :
    """
    p[0] = ''
    return p
```

```
def p_token1(p):
    """
    Token : Operation
           | Quickie
           | Misk
           | Manuver
           | Logic
           | Loop
           | Condition
           | Variables
           | Elem
           | Io
    """
    p[0] = p[1]
    return p
```

```
def p_operations1(p):
    """
    Operation : PLUS
    """
    p[0] = 'ADD\n'
    return p
```

```
def p_operations2(p):
    """
    Operation : TIMES
    """
    p[0] = "MUL\n"
    return p
```

```
def p_operations3(p):
    """
    Operation : DIVIDE
```

```

    """
    p[0] = "DIV\n"
    return p

def p_operations4(p):
    """
    Operation : MINUS
    """
    p[0] = "SUB\n"
    return p

def p_operations5(p):
    """
    Operation : EXP
    """
    v = parser.var_counter
    parser.var_counter += 2

    p[0] = "storeg " + str(v) + "\nDUP 1\nstoreg" + (str(v+1)) + "\nLOOPLABEL" + str(parser.loop_label)
    parser.loop_label += 1

    return p

def p_operations6(p):
    """
    Operation : MOD
    """
    p[0] = "MOD\n"
    return p

def p_operations7(p):
    """
    Operation : DIVMOD
    """
    v = parser.var_counter
    parser.var_counter += 1

    p[0] = "pushsp\nLOAD -1\npushsp\nLOAD -1\nMOD\nstoreg " + str(v) + "\nSWAP\npushg " + str(v)
    return p

def p_loop1(p):
    """
    Loop      : DO Options LOOP
    """
    v = parser.var_counter
    parser.var_counter += 2

    loop = "storeg " + str(v+1) + "\n" + "storeg " + str(v) + "\n" + "LOOPLABEL" + str(parser.loop_label)

```

```

    p[0] = handle_loop_var(loop, v+1)
    parser.loop_label += 1
    return p

def p_loop2(p):
    """
    Loop          : BEGIN Options UNTIL
    """

    p[0] = "LOOPLABEL" + str(parser.loop_label) + ":\n" + p[2] + "jz LOOPLABEL" + str(parser.loop_label)
    parser.loop_label += 1
    return p

def p_loop3(p):
    """
    Loop          : BEGIN Options WHILE Options REPEAT
    """

    p[0] = "LOOPLABEL" + str(parser.loop_label) + ":\n" + p[2] + "pushi 0\nEQUAL\nNOT\njz LOOPLABEL" + str(parser.loop_label)
    parser.loop_label += 2
    return p

def p_loop4(p):
    """
    Loop          : I
    """

    p[0] = "loopvar\n"
    return p

def p_loop5(p):
    """
    Loop          : DO Options PLUSLOOP
    """

    v = parser.var_counter
    parser.var_counter += 2

    loop = "storeg " + str(v+1) + "\n" + "storeg " + str(v) + "\n" + "LOOPLABEL" + str(parser.loop_label)

    p[0] = handle_loop_var(loop, v+1)
    parser.loop_label += 1
    return p

def p_condition1(p):
    """
    Condition     : IF Options ELSE Options THEN
    """

```

```

    c = parser.if_counter
    parser.if_counter += 1

    p[0] = "jz " + "ELSE" + str(c) + "\n" + p[2] + "jump ENDIF" + str(c) + "\n" + "ELSE" + str(c)
    return p

def p_condition2(p):
    """
    Condition    : IF Options THEN
    """
    c = parser.if_counter
    parser.if_counter += 1

    p[0] = "jz " + "ENDIF" + str(c) + "\n" + p[2] + "ENDIF" + str(c) + ":\n"
    return p

def p_quickie1(p):
    """
    Quickie      : QUICKADDONE
    """
    p[0] = "pushi 1\nADD\n"
    return p

def p_quickie2(p):
    """
    Quickie      : QUICKADDTWO
    """
    p[0] = "pushi 2\nADD\n"
    return p

def p_quickie3(p):
    """
    Quickie      : QUICKSUBONE
    """
    p[0] = "pushi 1\nSUB\n"
    return p

def p_quickie4(p):
    """
    Quickie      : QUICKSUBTWO
    """
    p[0] = "pushi 2\nSUB\n"
    return p

def p_quickie5(p):
    """
    Quickie      : QUICKMULTTWO
    """

```



```

    p[0] = "pushi 2\nMUL\n"
    return p

def p_quickie6(p):
    """
    Quickie      : QUICKDIVTWO
    """
    p[0] = "pushi 2\nDIV\n"
    return p

def p_logic1(p):
    """
    Logic        : EQUAL
    """
    p[0] = "EQUAL\n"
    return p

def p_logic2(p):
    """
    Logic        : NOTEQUAL
    """
    p[0] = "EQUAL\nNOT\n"
    return p

def p_logic3(p):
    """
    Logic        : GREATER
    """
    p[0] = "SUP\n"
    return p

def p_logic4(p):
    """
    Logic        : LESS
    """
    p[0] = "INF\n"
    return p

def p_logic5(p):
    """
    Logic        : ZEROEQUALS
    """
    p[0] = "pushi 0\nEQUAL\n"
    return p

def p_logic6(p):
    """
    Logic        : ZEROLESS

```

```

    """
    p[0] = "puchi 0\nINF\n"
    return p

def p_logic7(p):
    """
    Logic      : ZEROGREATER
    """
    p[0] = "pushi 0\nSUP\n"
    return p

def p_logic8(p):
    """
    Logic      : FALSE
    """
    p[0] = "pushi 0\n"
    return p

def p_logic9(p):
    """
    Logic      : TRUE
    """
    p[0] = "pushi 1\n"
    return p

def p_misk1(p):
    """
    Misk       : ABS
    """
    c = parser.if_counter
    parser.if_counter += 1

    p[0] = "DUP 1\npushi 0\nINF\njz " + "ENDIF" + str(c) + "\npushi -1\nMUL\nENDIF" + str(c) + "
    return p

def p_misk2(p):
    """
    Misk       : NEGATE
    """
    p[0] = "pushi -1\nMUL\n"
    return p

def p_misk3(p):
    """
    Misk       : MIN
    """
    c = parser.if_counter
    parser.if_counter += 1

```

```

p[0] = "pushsp\nLOAD -1\npushsp\nLOAD -1\nINF\njz " + "ELSE" + str(c) + "\nPOP 1\njump ENDIF"
return p

def p_misk4(p):
    """
    Misk          : MAX
    """
    c = parser.if_counter
    parser.if_counter += 1

    p[0] = "pushsp\nLOAD -1\npushsp\nLOAD -1\nSUP\njz " + "ELSE" + str(c) + "\nPOP 1\njump ENDIF"
    return p

def p_manuver1(p):
    """
    Manuver       : DUP
    """
    p[0] = "DUP 1\n"
    return p

def p_manuver2(p):
    """
    Manuver       : 2DUP
    """
    p[0] = "pushsp\nLOAD -1\npushsp\nLOAD -1\n"
    return p

def p_manuver3(p):
    """
    Manuver       : SWAP
    """
    p[0] = "SWAP\n"
    return p

def p_manuver4(p):
    """
    Manuver       : ROT
    """

    v = parser.var_counter
    parser.var_counter += 1

    p[0] = "storeg " + str(v) + "\nSWAP\npushg " + str(v) + "\nSWAP\n"
    return p

def p_manuver5(p):
    """

```

```

Manuver      : OVER
"""
p[0] = "pushsp\nload -1\n"
return p

def p_manuver6(p):
    """
    Manuver      : DROP
    """
    p[0] = "POP 1\n"
    return p

def p_variables1(p):
    """
    Variables      : VARIABLE ID
    """
    if p[2] in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(2)}, position {p.lexpos(2)}: Duplicat
        p[0] = '\n'
    else:
        parser.symbol_table[p[2]] = (parser.var_counter,1)
        parser.var_counter += 1
        p[0] = ""
    return p

def p_variables2(p):
    """
    Variables      : ID EXCLAMATION
    """
    if p[1] not in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(1)}, position {p.lexpos(1)}: Unknown
        p[0] = '\n'
    else:
        pos,t = parser.symbol_table[p[1]]
        p[0] = "storeg " + str(pos) + "\n"
    return p

def p_variables3(p):
    """
    Variables      : ID ATSIGN
    """
    if p[1] not in parser.symbol_table:
        parser.exito = False

```

```

        errors.append(f"\tError detected in line {p.lineno(1)}, position {p.lexpos(1)}: Unknown I
    p[0] = '\n'
else:
    pos,t = parser.symbol_table[p[1]]
    p[0] = "pushg " + str(pos) + "\n"

return p

def p_variables4(p):
    """
    Variables      : ID QUESTIONMARK
    """
    if p[1] not in parser.symbol_table:
        parser.exito = False
        errors.append(f"\tError detected in line {p.lineno(1)}, position {p.lexpos(1)}: Unknown I
        p[0] = '\n'
    else:
        pos,t = parser.symbol_table[p[1]]
        p[0] = "pushg " + str(pos) + "\n" + "writei\n"

return p

def p_variables5(p):
    """
    Variables      : ID
    """

    if macro.get(p[1]):
        m1, m2, m3 = macro[p[1]]
        m2 += 1
        if m2 == 1 or m3:
            m1_new = re_label(m1, m2, p[1])
            macro[p[1]] = (m1, m2, m3)
            p[0] = m1_new
        else:
            parser.exito = False
            parser.detected_error = 1
            errors.append(f"\tError detected in line {p.lineno(1)}, position {p.lexpos(1)}: Unknown I
            p[0] = '\n'
    return p

def p_io1(p):
    """
    Io              : DOT
    """
    p[0] = "writei\n"
    return p

```

```

def p_io2(p):
    """
    Io          : KEY
    """
    p[0] = "READ\nCHRCODE\n"
    return p

def p_io3(p):
    """
    Io          : CHAR
    """
    p[0] = f'pushs"{p[1]}"\nCHRCODE\n'
    return p

def p_io4(p):
    """
    Io          : CR
    """
    p[0] = "WRITELN\n"
    return p

def p_io5(p):
    """
    Io          : SPACE
    """
    p[0] = 'pushs" "\nwrites\n'
    return p

def p_io6(p):
    """
    Io          : SPACES
    """
    v = parser.var_counter
    parser.var_counter += 1
    space = 'pushs" "\nwrites\n'
    p[0] = "pushi 0\nSUB\n" + "storeg " + str(v) + "\n" + "LOOPLABEL" + str(parser.loop_label)
    parser.loop_label += 1
    return p

def p_io7(p):
    """
    Io          : EMIT
    """
    p[0] = "WRITECHR\n"
    return p

```

```

def p_elem1(p):
    """
    Elem          : NUMBER
    """
    p[0] = "pushi " + str(p[1]) + "\n"
    return p

def p_elem2(p):
    """
    Elem          : STRING
    """
    p[0] = "pushs" + ' ' + p[1] + '\n' + "writes\n"
    return p

def p_error(p):
    parser.exito = False
    errors.append(f"\tSyntactic error detected => Bad use of reserved words\n")

parser = yacc.yacc()
parser.exito = True
parser.if_counter = 1
parser.var_counter = 0
parser.loop_label = 0
parser.symbol_table = {} # id qual pointer tamanho
parser.called_func = {} # id de função - número atual de label utilizada

def main():
    if len(sys.argv) > 2:
        print("Uso: python analysyn.py")
        sys.exit(1)

    data = ''
    for line in sys.stdin:
        data += line

    result = parser.parse(data)
    if parser.exito:
        print("Parsing successful.")
        with open("target_file.txt", 'w') as target_file:
            target_file.write(result)
            print(f"Result written to target_file.txt")
    else:
        print("Parsing unsuccessful.")
        with open("error_file.txt", 'w') as error_file:
            out = f"{len(errors)} ERROR(S) FOUND:\n\n"
            for error in errors:
                out += error

```

```

        error_file.write(out)
        print(f"Result written to error_file.txt")

if len(sys.argv) != 2:
    print("Uso: python analysyn.py <input_file>")
    sys.exit(1)

input_file = sys.argv[1]
with open(input_file, 'r') as file:
    data = file.read()

result = parser.parse(data)
if parser.exito:
    print("Parsing successful.")
    with open("target_file.txt", 'w') as target_file:
        target_file.write(result)
        print(f"Result written to target_file.txt")
else:
    print("Parsing unsuccessful.")
    with open("error_file.txt", 'w') as error_file:
        out = f"{len(errors)} ERROR(S) FOUND:\n\n"
        for error in errors:
            out += error
        error_file.write(out)
        print(f"Result written to error_file.txt")
if len(sys.argv) != 2:
    print("Uso: python analysyn.py <input_file>")
    sys.exit(1)

input_file = sys.argv[1]
with open(input_file, 'r') as file:
    data = file.read()

result = parser.parse(data)
if parser.exito:
    print("Parsing successful.")
    with open("target_file.txt", 'w') as target_file:
        target_file.write(result)
        print(f"Result written to target_file.txt")
else:
    print("Parsing unsuccessful.")
    with open("error_file.txt", 'w') as error_file:
        out = f"{len(errors)} ERROR(S) FOUND:\n\n"
        for error in errors:
            out += error
        error_file.write(out)
        print(f"Result written to error_file.txt")

```



```
if __name__ == "__main__":  
    main()
```

Processamento de Linguagens

Engenharia Informática (3º ano)

Projeto Final

18 de Março de 2024

Objectivos e Organização

Este trabalho prático tem como principais objectivos:

- aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora;
- desenvolver um compilador gerando código para um objetivo específico;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

Na resolução dos trabalhos práticos desta UC, aprecia-se a imaginação/criatividade dos grupos em todo o processo de desenvolvimento! Deve entregar a sua solução até Domingo dia 12 de Maio.

O ficheiro com o relatório e a solução deve ter o nome 'pl2024- projeto-grNN.zip' de deverá estar no formato ZIP, em que NN corresponderá ao número de grupo. O número de grupo será ou foi atribuído no registo das equipas do projeto.

A submissão deverá ser feita no Black Board da UC.

Na defesa, a realizar na semana de 13 a 17 de Maio, o programa desenvolvido será apresentado aos membros da equipa docente, totalmente pronto e a funcionar (acompanhado do respectivo relatório de desenvolvimento) e será defendido por todos os elementos do grupo, em data e hora a marcar. O relatório a elaborar, deve ser claro e, além do respectivo enunciado, da descrição do problema, das decisões que lideraram o desenho da solução e sua implementação, deverá conter exemplos de utilização (textos fontes diversos e respectivo resultado produzido). Como é de tradição, o relatório será escrito em LaTeX.

Recursos e documentação

Estão disponíveis na internet vários recursos quer de documentação quer de ambientes de programação para Forth que o aluno poderá usar para estudar a linguagem e perceber melhor o problema.

Podemos destacar os seguintes recursos que foram usados na preparação do projeto e para criar os programas exemplo neste enunciado:

- **Documentação:** [um dos muito manuais online](#) apadrinhado pelo criador da linguagem: *"...I hope this book is not so easy and enjoyable that it seems trivial. Be warned that there is heavy content here*

and that you can learn much about computers and compilers as well as about programming." — Charles Moore;

- **IDE de desenvolvimento e testes:** [IDE para escrita e teste de programas](#), neste IDE os programas são compilados;
- **Máquina Virtual (VM)** a ser usada para a geração de código: [Documentação e IDE para testes e execução](#).

Forth: uma linguagem de programação

O Forth é uma linguagem de programação de baixo nível, baseada numa stack, que foi criada por Charles H. Moore na década de 1960. É conhecida pela sua simplicidade e eficiência, sendo amplamente utilizada em sistemas embebidos, controlo de hardware, sistemas operativos e outras aplicações que requerem desempenho e compactação.

A linguagem Forth tem como principais características:

- **Stack:** Todas as operações em Forth são realizadas utilizando uma stack de dados. Os valores são empilhados e desempilhados para execução das operações;
- **Notação pós-fixa (RPN - Reverse Polish Notation):** Forth utiliza a notação pós-fix, onde os operadores são colocados após os seus operandos. Por exemplo, 2 3 + realiza a operação de adição entre 2 e 3;
- **Extensível:** Forth é uma linguagem extremamente extensível, permitindo que novas palavras (ou funções) sejam definidas pelo utilizador de forma rápida e fácil. Isso permite uma grande flexibilidade na criação de programas específicos para determinadas aplicações;
- **Interpretada ou compilada:** Forth pode ser interpretada diretamente a partir do código fonte ou compilada para código nativo, dependendo da implementação. No nosso caso, iremos compilá-la;
- **Eficiência e compactação:** Forth é conhecida pela sua eficiência e compactação de código. Isso a torna uma escolha popular para sistemas com recursos limitados de hardware.

Em resumo, Forth é uma linguagem de programação simples, eficiente e flexível, que se destaca pela sua abordagem baseada numa stack e pela capacidade de se adaptar a uma ampla gama de aplicações, desde sistemas embebidos até sistemas operativos.

Existem muitos manuais disponíveis na internet pelo que não descreveremos mais da linguagem neste documento. No entanto, apresentam-se uma série de programas comentados que poderão ajudar a perceber a linguagem e poderão ser usados como teste no compilador.

Enunciado: o problema

Neste proejto, deverás implementar um compilador de Forth que deverá gerar código para a máquina virtual criada no contexto desta UC e [disponível online](#).

Ambas as linguagens, o Forth e o código máquina da máquina virtual serão apresentados e trabalhados nas aulas teóricas.

Na introdução deste documento, tens as localizações da documentação e ambientes de teste e desenvolvimento que terás de usar.

Este projeto será avaliado pelo seu grau de completude e apresentam-se já os seguintes níveis, do mais básico para o mais completo:

1. Suporte a todas as expressões aritméticas: soma, adição, subtração, divisão, resto da divisão inteira;
2. Suporte à criação de funções;
3. Suporte ao `print` de caracteres e strings (`.`, `." string", emit, char`);
4. Suporte a condicionais;
5. Suporte a ciclos;
6. Suporte a variáveis;
7. ...

Sintaxe

Um programa em FORTH é uma lista de palavras (`words`) separadas por espaço:

`WAKE.UP EAT.BREAKFAST WORK EAT.DINNER PLAY SLEEP`

ou

`." #S SWAP ! @ ACCEPT . *`

Baseada em Stacks

Todas as operações em FORTH assentam na manipulação de Stacks. A mais simples e mais usada é a Stack de Dados (`dataStack`).

A Stack de Dados está inicialmente vazia. Para colocarmos valores na stack, basta introduzirmos esses valores como palavras:

```
17 34 23
```

Para imprimir o valor no topo da stack usamos a palavra `.`:

```
< 17 34 23
< .
> 23
```

E agora usa os recursos apontados para estudares a linguagem.

Aritmética

Exemplo:

```
2 3 + .
2 3 + 10 + .
```

As expressões aritméticas escrevem-se no formato RPN ("Reverse Polish Notation").

```
30 5 - . ( 25=30-5 )
30 5 / . ( 6=30/5 )
30 5 * . ( 150=30*5 )
30 5 + 7 / . \ 5=(30+5)/7
```

Atalhos: 1+ 1- 2+ 2- 2* 2/

Experimentar no interpretador online:

```
10 1- .
7 2* 1+ . ( 15=7*2+1 )
```

Exercício: Conversão de expressões em formato infixo para pós-fixado

Escreve expressões FORTH para as seguintes expressões aritméticas:

```
( 12 * ( 20 - 17 ) )
( 1 - ( 4 * (-18) / 6 ) )
( 6 * 13 ) - ( 4 * 2 * 7 )
```

Definição de uma nova palavra ou função

Utilizam-se duas novas palavras: `:` e `;` Que marcam o início e o fim de uma definição de uma nova palavra ou função.

Exemplo: `: AVERAGE (a b -- avg) + 2/ ;`

```
: AVERAGE ( a b -- avg ) + 2/ ;
10 20 AVERAGE .
```

Input e Output de caracteres

```
CHAR W .
CHAR % DUP . EMIT
CHAR A DUP .
32 + EMIT
```

CHAR (<char> -- char , get ASCII value of a character)

Strings

```
: TOFU ." Yummy bean curd!" ;
TOFU
```

```
: SPROUTS ." Miniature vegetables." ;
: MENU
  CR TOFU CR SPROUTS CR
;
MENU
```

Input

```
: TESTKEY ( -- )
  ." Hit a key: " KEY CR
  ." That = " . CR
;
TESTKEY
```

Resumindo:

```
EMIT ( char -- , output character )
KEY ( -- char , input character )
SPACE ( -- , output a space )
SPACES ( n -- , output n spaces )
CHAR ( <char> -- char , convert to ASCII )
CR ( -- , start new line , carriage return )
." ( -- , output " delimited text )
```

Alguns programas exemplo em Forth

A seguir apresentam-se alguns programas em Forth que poderão ajudar a compreender melhor a linguagem e que podem ser usados para testar o compilador desenvolvido.

Hello world!

```
: hello-world ( -- )
  ." Hello, World!" cr ;

hello-world \ Call the defined word
```

Notas:

- `:` `hello-world` inicia a definição de uma nova palavra designada `hello-world`;
- `(--)` indica que `hello-world` não recebe argumentos e não deixa nada na stack;
- `." Hello, World!"` coloca na saída (stdout) a string `"Hello, World!"` sem alterar o estado da stack;
- `cr` escreve um carácter de mudança de linha `'\n'` na saída;
- `;` termina a definição da nova palavra.

Maior de 2 números passados como argumento

```
: maior2 2dup > if swap . ." é o maior " else . ." é o maior " then ;
77 156 maior2
```

Maior de 3 números passados como argumento

```
: maior2 2dup > if swap then ;
: maior3 maior2 maior2 . ;
2 11 3 maior3
```

Maior de N números passados como argumento

```
: maior2 2dup > if drop else swap drop then ;
: maior3 maior2 maior2 ;
: maiorN depth 1 do maior2 loop ;
2 11 3 4 45 8 19 maiorN .
```

Somatório de 1 até n-1

```
: somatorio 0 swap 1 do i + loop ;
11 somatorio .
```

Playing with chars and strings

```
( May the Forth be with you)
: STAR 42 EMIT ;
: STARS 0 DO STAR LOOP ;
: MARGIN CR 30 SPACES ;
: BLIP MARGIN STAR ;
: IOI MARGIN STAR 3 SPACES STAR ;
```

```

: IIO MARGIN STAR STAR 3 SPACES ;
: OIO MARGIN 2 SPACES STAR 2 SPACES ;
: BAR MARGIN 5 STARS ;
: F BAR BLIP BAR BLIP BLIP CR ;
: O BAR IOI IOI IOI BAR CR ;
: R BAR IOI BAR IIO IOI CR ;
: T BAR OIO OIO OIO OIO CR ;
: H IOI IOI BAR IOI IOI CR ;
F O R T H

```

Fatorial

```

: factorial ( n -- n! )
  dup 0 = if
    drop 1
  else
    dup 1 - recurse *
  then ;

\ Example usage:
5 factorial . \ Calculate factorial of 5 and print result

```

Notas:

Explanation:

- `: factorial` inicia a definição de uma nova palavra designada `factorial`;
- `(n -- n!)` indica que `factorial` recebe um argumento (n) e deixa um resultado (n!) na stack;
- `dup 0 = if` testa se o argumento é 0;
- Se o argumento for 0, este é descartado e o valor 1 é colocado na stack (será o caso de base para o cálculo do fatorial);
- Se o argumento não for 0, este é duplicado, subtrai-se 1 ao duplicado, a função é chamada recursivamente, e o seu resultado é multiplicado pelo argumento original;
- `then` marca o fim do bloco condicional;
- `\ Example usage:` é um comentário que indica como usar a palavra/função nova;
- `5 factorial .` calcula o fatorial de 5 e escreve o resultado na saída.

Este programa mostra a utilização da recursividade em Forth, neste caso, para calcular o fatorial.

Somatório de n números

```

: sum ( n -- sum )
  0 swap 1 do
    i +
  loop ;

\ Example usage:
5 sum .

```


Notas:

- `: sum` inicia a definição de uma nova palavra designada `sum`;
- `(n -- sum)` indica que `sum` recebe 1 argumento (`n`) e deixa 1 resultado (`sum`) na stack;
- `0 swap` inicializa `sum` a 0 e troca o argumento com o topo da stack;
- `1 do` inicia o ciclo de 1 até ao valor do argumento;
- `i +` adiciona o índice do ciclo corrente (`i`) a `sum`;
- `loop` termina o ciclo;
- `\ Example usage:` é um comentário;
- `5 sum .` calcula e imprime o somatório dos números inteiros de 1 a 5.